Vers la vérification de SMALA, un langage réactif interactif

Nicolas Nalpon, Célia Picard, Cyril Allignol, Sébastien Leriche

Vers la vérification de SMALA, un langage réactif interactif

Nicolas Nalpon, Celia Picard, Cyril Allignol et Sébastien Leriche

ENAC, Université de Toulouse - France

Résumé

SMALA est un langage réactif dédié à la programmation de systèmes à forte composante interactive. Pour pouvoir utiliser ce langage dans un contexte critique, il est nécessaire de pouvoir apporter des garanties. Ainsi, nous nous intéressons à la vérification formelle du compilateur de SMALA, avec pour objectif de garantir la préservation de la sémantique du programme source à la compilation. Dans un premier temps, nous avons limité notre étude à un sous-ensemble de SMALA. Dans cet article, nous présentons une version préliminaire de la sémantique opérationnelle de ce sous-ensemble.

Mots-clés : compilation certifiée, langage réactif-interactif, sémantique opérationnelle

1 Introduction

Bon nombre de systèmes critiques comportent une importante composante interactive, c'est-à-dire que des utilisateurs manipulent le système qui doit pouvoir réagir à leurs actions et afficher des informations. C'est notamment le cas de nombreux systèmes aéronautiques, tels que les cockpits d'avions, d'hélicoptères ou les systèmes de contrôle aérien.

Un système qui réagit à des actions issues de son environnement est dit réactif. La programmation réactive est un paradigme qui s'intéresse à la gestion des flux de données dans ces systèmes [2]. Néanmoins, les langages de programmation classiques, même réactifs ou event-driven, sont peu adaptés à la conception de logiciels interactifs. En effet, ils ne permettent pas une intégration fluide et unifiée des différentes phases de conception (design graphique, architecture, codage...) et plus particulièrement de la partie interactive. C'est cette unification des paradigmes réactif et interactif que vise SMALA [5]. Ce langage, dédié à la conception de programmes interactifs, en particulier aéronautiques, a ainsi été utilisé pour développer le tableau de bord de l'hélicoptère électrique Volta [1].

Les logiciels critiques avioniques doivent se conformer à la norme DO-178C, dont le supplément DO-333 encourage à utiliser les méthodes formelles pour la vérification. En particulier, il est ainsi intéressant de certifier le compilateur utilisé pour créer le logiciel. Cela consiste à démontrer formellement que la compilation n'introduit pas d'erreur dans le code généré, i. e., s'assurer de la préservation de la sémantique du programme source dans le code généré. Aujourd'hui, cette vérification est le plus souvent manuelle [4].

Nous visons à vérifier un compilateur pour SMALA, grâce à l'assistant de preuve Coq. Dans la littérature, divers travaux s'intéressent à la compilation certifiée. Ainsi, Compcert [4] est un compilateur opérationnel du langage C vérifié avec Coq ; CakeML [6] est

un projet de vérification d'un sous ensemble de StandardML avec l'assistant de preuve HOL4 ; Vélus [3] est un compilateur vérifié pour le langage réactif synchrone Lustre avec Coq. CakeML et Vélus sont en cours de développement mais déjà partiellement opérationnels. Ces trois travaux traitent de langages ayant des paradigmes de programmation différents les uns des autres. Le paradigme réactif synchrone de Lustre est celui qui se rapproche le plus du paradigme réactif de SMALA. Néanmoins, le paradigme synchrone, bien que conçu pour les systèmes réactifs, est limitant car il impose de fortes contraintes sur le temps de réaction (par exemple, la synchronisation des flux d'entrée du programme qui doivent tous donner leurs valeurs en même temps) et sur les ressources utilisées pour le développement du programme (la mémoire utilisée est bornée) [3]. SMALA est plus souple, n'imposant aucune contrainte sur le temps de réaction ni sur les ressources utilisées et apporte des aspects interactifs absents de Lustre. C'est un langage spécifique, défini via de nombreuses bibliothèques C++ dont une large bibliothèque graphique. Pour faciliter la certification du compilateur, nous visons à remplacer la plupart de ces bibliothèques C++ par des bibliothèques développées directement en SMALA. Nous avons commencé par définir un sous-ensemble très réduit que nous compléterons petit à petit, selon le besoin. Pour cela nous nous sommes affranchis de la partie interactive de Smala. Dans un premier temps, nous nous intéressons seulement au côté réactif du langage.

La section 2 présente SMALA et le sous-ensemble considéré dans cet article, qui permet d'en étudier les aspects réactifs. La section 3 développe une première formalisation de la sémantique de ce sous-ensemble. Enfin, la section 4 présente des pistes pour la suite.

2 SMALA

Le concept fondamental au cœur de SMALA est celui de processus. Un processus est une entité qui a sa propre sémantique d'exécution et un état d'activation qui peut être "activé" (le processus peut s'exécuter) ou "désactivé" (le processus attend d'être activé). Parmi ces processus, nous en présentons ici quatre qui sont essentiels et qui constituent notre sous-ensemble minimal. Le couplage (ou *binding*, noté $\rightarrow$) associe un processus source à un processus cible. Lorsqu'un processus source change d'état, il active les processus cibles associés. La *property* encapsule une valeur. L'*assignment* $=:$ permet de copier la valeur d'une *property* dans une autre. Enfin, le *connector* $\Rightarrow$ permet, à chaque activation de la *property* source, de copier sa valeur dans la *property* cible et de propager son activation. Afin d'avoir une meilleure compréhension du langage, nous en détaillons un exemple qui couvre tout le sous-ensemble présenté ici.

```
_main_
Frame f ("titre", 0, 0, 500, 400) // Création d'une fenêtre
Rectangle r (0, 0, 50, 99, 0, 0) // Création d'un rectangle
Int height (500) // Définition d'une Int Property
Exit quit (0, 1) // Processus pour quitter l'application
// L'activation de l'attribut close de f est propagée à
   quit, i.e.  fermer la fenêtre quitte l'application
f.close -> quit
// Copie la valeur de height dans r.height seulement à
   l'initialisation de l'application
height =: r.height
// La modification de la largeur de la fenêtre provoque la
   copie de cette valeur dans la largeur du rectangle
f.width => r.width
```

Un programme SMALA a une phase d'initialisation durant laquelle les processus sont déclarés et instanciés et une phase d'exécution pendant laquelle il attend un signal, provoqué par un évènement, pour déclencher une action (ci-dessus, la fermeture de la fenêtre).

3 Sémantique

La sémantique opérationnelle de SMALA reflète les phases d'initialisation et d'exécution précédemment évoquées.

3.1 Initialisation

La phase d'initialisation construit l'environnement E et une fonction S à partir du programme Ins. E associe à chaque *property* sa valeur. $E \vdash e \Downarrow v$ est l'évaluation d'une expression e en une valeur v dans E. $E, S \vdash i \rightsquigarrow E', S'$ dénote un changement d'état qui consiste en la modification de E ou S. Les règles d'initialisation sont les suivantes :

$$\text{Init}\ \frac{E, S \vdash i \rightsquigarrow E', S' \quad E', S' \vdash Ins \rightsquigarrow E'', S''}{E, S \vdash i :: Ins \rightsquigarrow E'', S''} \qquad \text{InitConnector}\ \frac{}{E, S \vdash e \Rightarrow x \rightsquigarrow E, S'} \ \text{avec } S' = \{(y, e \Rightarrow x) \mid y \in FV(e)\} \cup S$$

$$\text{EndInit}\ \frac{}{E, S \vdash [] \rightsquigarrow E, S} \qquad \text{InitProperty}\ \frac{E \vdash e \Downarrow v}{E, S \vdash Int\ x\ (e) \rightsquigarrow E[x/v], S}$$

La règle Init parcourt Ins et applique à chaque instruction la bonne règle d'initialisation.

Nous détaillons seulement les règles d'initialisation pour certains composants. Les autres règles se déduisent aisément de celles données. $E[x/v]$ associe à x la valeur v.

- La règle InitProperty exprime qu'une *property* s'évalue à l'initialisation et qu'elle n'est plus utilisée par la suite, i. e., l'instruction n'est pas ajoutée à S. $Int\ x\ (e)$ dénote la déclaration d'une *property* x avec pour valeur l'expression e.
- La règle InitConnector met à jour la fonction S en ajoutant $e \Rightarrow x$ à l'image des variables libres de e. Ainsi, l'activation d'une variable libre de e active le *connector*.

3.2 Exécution

La phase d'exécution regroupe deux ensembles de règles aux fonctionnalités différentes.

3.2.1 Règles d'exécution

Les règles d'exécution ci-dessous donnent la sémantique des processus du langage. Nous commentons ici uniquement la règle ExecConnector : l'expression e s'évalue en v dans E puis la valeur de x dans E est mise à jour en v.

$$\text{ExecBinding}\ \frac{E \vdash x_2 \rightsquigarrow E'}{E \vdash x_1 \rightarrow x_2 \rightsquigarrow E'} \qquad \text{ExecConnector}\ \frac{E \vdash e \Downarrow v}{E \vdash e \Rightarrow x \rightsquigarrow E[x/v]}$$

$$\text{ExecAssignment}\ \frac{E \vdash e \Downarrow v}{E \vdash e =: x \rightsquigarrow E[x/v]} \qquad \text{ExecProperty}\ \frac{}{E \vdash x \rightsquigarrow E}$$

3.2.2 Règles de propagation

Les règles de propagation ci-dessous définissent l'exécution d'un programme à partir du moment où celui-ci reçoit un signal.

- La règle PropagSig attend qu'un signal issu d'un évènement externe ou interne mette à jour une variable correspondant à un processus dans E. Elle exécute alors toutes les instructions déclenchées par le processus en question. La fonction S renvoie un ensemble d'instructions $S(s)$ qui constitue la file d'ensembles d'instructions I. Elle sert à générer un ordre d'exécution (les ensembles d'instructions sont ordonnés entre eux, mais les instructions au sein d'un même ensemble ne le sont pas)
- La règle PropagConnector exprime que la file d'ensembles d'instructions I complétée par l'ensemble des instructions activables par x est évaluée dans l'environnement résultant de la règle ExecConnector

$$\text{PropagSig}\ \frac{\exists s \in E,\ E[s/v], S \vdash [S(s)] \rightsquigarrow E', S}{E, S \vdash [] \rightsquigarrow E', S}$$

$$\text{PropagConnector}\ \frac{E \vdash e \Rightarrow x \rightsquigarrow E' \qquad E', S \vdash (ins :: I)@[S(x)] \rightsquigarrow E'', S}{E, S \vdash (\{e \Rightarrow x\} \cup ins) :: I \rightsquigarrow E'', S}$$

4 Suite des travaux

Cet article présente une première sémantique d'un sous-ensemble de SMALA, un langage réactif dédié à la conception de systèmes interactifs. Ce sous-ensemble n'inclut pas pour l'instant d'éléments liés à la partie interactive. Il amorce de nombreuses pistes que nous explorerons dans nos travaux futurs. Nous ajouterons au sous-ensemble les expressions, en particulier les opérations binaires, pour vérifier que notre sémantique ne génère pas d'incohérences lors de la propagation des changements (glitch) [2]. Nous formaliserons l'activation et la désactivation des processus. Enfin, nous formaliserons cette sémantique dans Coq et prouverons l'équivalence sémantique de compositions de composants.
Les différents travaux cités nous permettront tout d'abord de tester la robustesse de la sémantique et enrichir notre sous-ensemble de Smala. Par la suite, nous compilerons ce sous-ensemble en un langage impératif à définir et prouverons la correction de la compilation en s'inspirant des méthodes utilisées dans les projets CompCert, Vélus et CakeML.

References

[1] P. Antoine and S. Conversy. "Volta: the first all-electric conventional helicopter". In: *Proceedings of the More Electrical Aircraft Conference (MEA '17)* (2017).

[2] E. Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Computing Surveys* (2013).

[3] T. Bourke et al. "A Formally Verified Compiler for Lustre". In: *PLDI* (2017).

[4] X. Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assitant". In: *POPL* (2006).

[5] M. Magnaudet et al. "Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming". In: *ACM Hum-Comput. Interact* (2018).

[6] Y K. Tan et al. "A New Verified Compiler Backend for CakeMl". In: *ICFP* (2016).